Scheme in 4 days

A Navy/FNMOC presentation

Welcome!

Course material & Tools

R5RS document: terse.

Teach Yourself Scheme in Fixnum Days by Dorai Sitaram: direct.

Implementations: Gambit, DrScheme.

Emacs as editor.

Design Principles

Small, clear, fundamental concepts of computation.

Simple prefix syntax (functional composition):

`(- 2 3), (f 2 3), (- (* 3 4) (* 8 7))`

$\Rightarrow$ no precedence of operators.

Avoids: What does '`A[a--] = a--;`' mean?

Independent from computer binary representation.

Proper tail recursion.

Dynamic types, dynamic memory management.

Dynamic approach

Data types: vector, list, numbers, string, symbol, char, boolean, port, procedure (aka closure, function, continuation).

Dynamic types: no type declarations.

```
(define (f x y)  (list x y))
```

`x` and `y` have no static type, they may be bound to any data type.

`(f 1 2)` $\Rightarrow$ `(1 2)`

`(f "a" 3.4)` $\Rightarrow$ `("a" 3.4)`

Dynamic memory allocation

Vectors, lists, closures, etc., are allocated automatically in memory.

This dynamic space is the *heap*.

It is impossible to corrupt the heap by programming mistakes.

It is a "safe" language where the underlying machine is hidden.

Generic functions

The dynamic types have one advantage:

Generic functions can easily be built.

In general, Scheme compilers cannot catch type errors.

A programmer should describe variable types in comments.

Prototyping

Scheme is a good language for prototyping.

Data types do not have to be crafted in details early on.

Easier to modify in later stages of the life cycle of the software.

Standard?

There are two: IEEE $\subset$ R5RS.

For most Schemers, R5RS is the standard.

The IEEE promotes Scheme outside academia.

Will there be a R6RS? May be not.

But ...

Srfi, **S**cheme **R**equest **F**or **I**mplementation, is the new way.

Scheme implementations

Gambit: good debugger, good compiler.

DrScheme: good libraries, GUI environment.

Bigloo: good compiler, good tools, fast code, nice emacs environment (the BEE).

Chez Scheme: commercial.

Others: MIT Scheme, Scheme 48, ...

Most are free.

A good text editor

Emacs: Scheme mode, colors, parentheses.

Run Scheme interpreter in one buffer.

It is fast and simple.

You will not find ___ in R5RS

Threads, module system, records, detailed OS interface.

But, implementations do provide some of these, and much more in some cases.

Basic data types (1)

Numbers: `2`, `4.5`, `2+6i`, `3/4`, `1876281728671222`.

Char: `#\C`, `#\(`, `#\space`.

Boolean: `#t`, `#f`.

String: "Paris electric".

Basic data types (2)

Literal symbols, lists and vectors must be quoted.

`'(1 2 3)` otherwise `(1 2 3)` is a function call.

Symbol: `'products`, `'<=`, `'<xml>`, `'<?doc?>`.

List: `'(1 2 3)`, `'("al" al 2.3)`, `'(1 (2 3) (4))`.

Vector: `'#(2 3)`, `'#(left right 3.2)`

Pair: `'(a . 2)`, `'(2 . 2)`, `'(#\a . #\b)`.

Basic data type: port

`(current-output-port)` ⇒ `#<output-port>`

`(open-input-file "ttt")` ⇒ `#<input-port>`

We can read or write to ports.

```
(read port)
```

```
(display exp port)
```

```
(write exp port)
```

Basic data type: procedure

Procedure: `(lambda (x y) (- x y))`.

The procedure type represents functions and closures.

More on closures later.

Quote

'pay is the literal symbol pay

but

pay denotes the value of a variable.

We can also write (`quote pay`).

It is redundant to quote numbers, chars, booleans and strings.

Simple if

```
(if (< x 7)
    (display "That is small!")
    (display "Not so small"))
```

Let `L` = `(1 2 3)`

`(if L 'ok 'empty)` ⇒ `ok`

Anything not `#f` is true.

set!

set! is the assignment operator.

```
(set! x 2)
```

```
(set! L '(a b c))
```

x ⇒ 2

L ⇒ (a b c)

It can be used on local and global variables.

Function composition

Scheme syntax uses function composition.

$(f\ (g\ x))$

f and g are composed together.

```
(display (abs (- (* x y) z)))
```

Also called prefix notation.

Macros

Scheme macros are unlike C macros.

C macros use text processing.

Scheme macros uses structure forms.

```
(define-syntax and
   (syntax-rules ()
     ((and) #t)
     ((and e)  e)
     ((and e1 e2 ...)
      (if e1 (and e2 ...) #f))))
```

Unlike a function, a macro does not evaluate its argument.

The end?

Scheme is built from:

`set!`, `if`, `lambda`, composition of functions, data types, macros, and primitives.

The rest of Scheme is defined from these.

Having more syntactic forms is very useful.

So you have: `let`, `let*`, `cond`, `case`, `define`, etc.

Let us define ...

A function

```
(define f
  (lambda (x)
    (if (< x 0)
        (sqrt (abs x))
        (sqrt x))))
```

`(f -16)` $\Rightarrow$ 4

`(f 16)` $\Rightarrow$ 4

x is local to the lambda; it is a parameter.

f can be considered a variable of type function.

So, `define` may create variables of other types.

Define variables

`define` may introduce variables of any type:

```
(define l '(a b c d))

(define pi 3.141592653589793)

(define msg1 "Error in ...")

(define v1 '#(a b c d e))
```

Define, function special syntax

Functions may be defined using a special syntax:

```
(define (f x y)
   (+ x y))
```

is equivalent to

```
(define f
   (lambda (x y)
     (+ x y)))
```

A program = definitions + expressions

A Scheme program is a series of definitions and expressions.

```
(define x ...)
...
(define f (lambda (a b c) ...))
...
(define g (lambda (m n) ...))
...
(f 1 2 3)
```

A simple program

```
;; Only two expressions

(display "Hello world")
(newline)
```

A Scheme program is a sequence of expressions, some of which could be function or variable definitions.

Load

A large system can be split in several files.

The primitive ‘load’ dynamically load a Scheme file into the interpreter.

```
(load "partA")
(load "partB")
...
(load "partZ")
```

Global vs local

All definitions at "top-level" are global.

There are ways to define locals.

The macro `let` binds variables to values and evaluates a series of expressions.

```
(let ((x 2)
      (y 3))
  (+ x y))
```

`x` and `y` are local variables.

The value of `let` is the value of the last expression in its body.

Local function definitions

`define` may contain local functions.

In a function:

```
(define (f a b)

  (define (c n) (* n n))

  (sqrt (+ (c a) (c b))))
```

This is simply a useful syntax as a `let` could define the function `c`.

Local definitions in let

A `let` may have local functions.

```
(let ((x #f)
      (y #f))
  (define (reset)
     (set! x 0) (set! y 0))
  ...
  ... (reset) ...
  )
```

`reset` is local to the `let`, not accessible outside of it.

Numbers

Scheme has several number types: integer, complex, rational, real.

And several basic arithmetic and trigonometric functions: `+`, `*`, `-`, `/`, `expt`, `remainder`, `modulo`, `ceiling`, `floor`, `abs`, `max`, `min`, `sqrt`, `sin`, `cos`, `tan`, `gcd`, `lcm`, `acos`, `asin`, etc.

Exactness

Numbers are either exact or inexact.

`(exact? 2)` $\Rightarrow$ `#t`

`(inexact? 1.2)` $\Rightarrow$ `#t`

We always have

`(exact? x)` $\equiv$ `(not (inexact? x))`.

Integers and rationals are exact.

Reals are inexact.

Complex numbers are inexact or exact.

Bignum

Large integers can be used.

```
(expt 2 100)
```

$\Rightarrow$

```
1267650600228229401496703205376
```

Most Scheme implementations have bignums.

Inexact numbers

Inexact numbers are usually implemented using 64 bits IEEE-754 standard.

```
(sqrt 2)
```

$\Rightarrow$ `1.4142135623730951`

Scientific notation can be used: `1.3E-10`

Rational Numbers

Rationals are fractions: `(/ 1 3)` or `1/3`.

This is not `0.3333333` which is inexact.

All arithmetic operations can be applied to rationals.

`(+ 1/3 2/4)` $\Rightarrow$ `5/6`

`(/ 4/5 9/7)` $\Rightarrow$ `28/45`

`(* 4/5 9/7 4/7 9/2)` $\Rightarrow$ `648/245`

They are exact numbers as are integers.

`(exact? 1/3)` $\Rightarrow$ **`#t`**

Complex numbers

Complex numbers: 2+4i, -i, +i, 2.3+8.9i

All arithmetic, algebraic, and trigonometric functions can be applied to them.

The can be read or written using `read`, `display` or `write`.

They can be exact or inexact.

`(+ 2+4i 4+8i)` ⇒ `6+12i`

`(+ 2.32+4.5i 5.6+i)` ⇒ `7.92+5.5i`

`(exact? (+ 2+4i 4+8i))` ⇒ `#t`

`(exact? (+ 2.32+4.5i 5.6+i))` ⇒ `#f`

Char

A constant char uses the `#\` syntax: `#\a`, `#\0`, etc.

R5RS does not impose a coding standard.

Most implementation assumes ASCII 7 bits.

`(integer->char 65)` $\Rightarrow$ `#\A`

`(char->integer #\A)` $\Rightarrow$ 65

R5RS imposes: all letter and digit codes are in order.

Some implementations have Unicode.

String, 1

A constant string is double-quoted.

"Monterey, California."

`string-length` returns the length of a string.

`(string-ref s i)` refers to the ith character of s.

`(string-set! s i v)` modifies the ith character of s.

String, 2

`string-append` concatenates strings.

(string-append "Hello " "world") $\Rightarrow$ "Hello world"

`(make-string 10 #\.)` $\Rightarrow$ ".........."

`(substring "abcdef" 1 3)` $\Rightarrow$ "bc"

substring creates a new string.

List

The empty list is denoted `'()`.

They may contain heterogenous data types

```
'(#\a a_symbol 2.3 "is")
```

The function `list` creates a list.

`(list 1 2 3)` ⇒ `(1 2 3)`

Subtlety: `'(1 2 3)` is not exactly the same as `(list 1 2 3)` since `list` creates new pairs.

List and pairs

Lists are constructed from pairs:

`(cons 1 2)` $\Rightarrow$ `(1 . 2)`

`(list 1 2 3)` is equivalent to

```
(cons 1 (cons 2 (cons 3 '())))
```

Lists are allocated in the heap.

List, car, cdr

The most common operations on lists:

`(car '(1 2 3))` $\Rightarrow$ 1

`(cdr '(1 2 3))` $\Rightarrow$ (2 3)

`(cdr '())` $\Rightarrow$ error

`(car '())` $\Rightarrow$ error

List, c..r

`(cadr '(1 2 3))` $\Rightarrow$ `2`

`(cddr '(1 2 3))` $\Rightarrow$ `(3)`

In general cαr where α is a series of at most four 'a's and 'd's.

`(caadr x)` $=$ `(car (car (cdr x)))`

List?, pair?, null?

`(null? L)` is `#t` iff `L` is the empty list.

`(pair? L)` is `#t` iff `L` is a pair.

`(list? L)` is `#t` iff `L` is a proper list.

`(list? '(1 2 3))` $\Rightarrow$ `#t`

`(list? '(1 2 . 3))` $\Rightarrow$ `#f`

`Pair?` should be fast, but `list?` may be slow.

Length, reverse, append

Three common basic functions:

`(length '(a b c))` $\Rightarrow$ 3.

`(append '(1 2) '(a b))` $\Rightarrow$ `(1 2 a b)`

`(reverse '(a b c d))` $\Rightarrow$ `(d c b a)`

Append copies all its arguments, except the last one.

Length, a peek at recursion

Here is a possible recusive definition of length:

```
(define (length L)
  (if (pair? L)
      (+ 1 (length (cdr L)))
      0))
```

But it is slightly incorrect as it should return an error if L is not a proper list.

List, list-ref

`(list-ref '(1 2 3 4) 3)` $\Rightarrow$ 4

`(list-ref L i)` extracts the ith element of L.

A peek at a recursive definition.

```
(define (my-list-ref l i)
  (if (pair? l)
      (if (= i 0)
          (car l)
          (my-list-ref (cdr l) (- i 1)))
      (error "list-ref, not a pair.")))
```

Circular lists

Circular lists may be constructed.

`(set! L '(a b c))`

`(set-cdr! L L)`

L $\Rightarrow$ `(a a a a a ...`

It has infinite length.

`(length L)` does not terminate.

`(append l '(z))` does not terminate.

Graph

General graphs may be constructed from lists.

Some implementations support reading and writing graphs. (circular references)

Equivalences, equal?

equal? is the most general (vector, list, etc.)

`(equal? '(1 2 3) '(1 2 3 ))` $\Rightarrow$ `#t`

`(equal? '#(a b c) '#(a b c))` $\Rightarrow$ `#t`

`(equal? 1/4 2/8)` $\Rightarrow$ `#t`

`(equal? 3 3)` $\Rightarrow$ `#t`

`(equal? "allo" "allo")` $\Rightarrow$ `#t`

Equivalences, eqv?

`eqv?` can compare "unstructured" values.

`(eqv? x y)` implies `(equal? x y)`

The inverse implication is not true.

`(eqv? 2 2)` $\Rightarrow$ `#t(eqv? #\a #\b)` $\Rightarrow$ `#f`

`(eqv? 'aa 'aa)` $\Rightarrow$ `#t`

`(eqv? "aa" "aa")` $\Rightarrow$ unspecified

Looking alike symbols are equal according to eqv?; not true for string.

Equivalences, eq?

eq? compare memory locations.

`(eq? (list 1 2 3) (list 1 2 3))` $\Rightarrow$ `#f`

Because `list` creates new pairs.

```
(define l '(1 2 3))
```

```
(set! x l)
```

`(eq? x l)` $\Rightarrow$ `#t`

Note: there is no such thing as arithmetic on pointers.

Searching through a list

There are three basic functions.

`(member e L)`

searches through `L` for `e`, using `equal?`.

It returns the sublist starting with `e` or `#f`.

`(member 2 '(1 2 3))` $\Rightarrow$ `(2 3)`

`(member 4 '(1 2 3))` $\Rightarrow$ `#f`.

`(memv e l)` uses `eqv?`

`(memq e l)` uses `eq?`

Association list, alist

A list of pairs:

```
(define al
  '((Canada . Ca) (France . Fr) (Quebec . Qc)))
```

`(assoc 'Canada al)` ⇒ `(Canada . Ca)`

`(assoc 'Suisse al)` ⇒ `#f`

`assoc` uses `equal?` to compare keys.

Also: `assq` using `eq?`, `assv` using `eqv?`.

MBL in Metcast

A MBL request is a list structure.

Let R be

```
(area
  (bounding-box 46 -177 20 100)
  (products (METAR)))
```

To have the bounding-box:

```
(assoc 'bounding-box (cdr R))
```

$\Rightarrow$ `(bounding-box 46 -177 20 100)`

assv would work too.

Vectors

Only one dimensional vectors.

They are heterogenous.

`'#("string" symbol 2.3)`

`vector` creates vectors in the heap.

`(vector 1 2 3 4)` $\Rightarrow$ `#(1 2 3 4)`

It is of variable arity.

`(vector-ref v i)` $\Rightarrow$ ith element of v.

`(vector-set! v i val)` sets ith element to val.

Data type conversions

(string->list "abc") ⇒ (#\a #\b #\c)

(list->string '(#\a #\b #\c)) ⇒ ”abc”

(list->vector '(1 2 a)) ⇒ #(1 2 a)

(vector->list '#(a 5 6)) ⇒ (a 5 6)

(symbol->string 'abc) ⇒ "abc"

(string->symbol "abc67<>") ⇒ abc67<>

Equivalences, according to type

`=` for numbers.

`(= 1/2 2/4)` ⇒ `#t`

`(= "aa" "aa")` ⇒ error

`char=?` for characters.

`(char=? #\a #\b)` ⇒ `#f`

`string=?` for strings.

`(string=? "ciao" "ciao")` ⇒ `#t`

Dynamic types 1

Variables do not have a static type.

Primitives to test expression types:

`(string? x)`, `(procedure? x)`, `(char? x)`, `(number? x)`, `(real? x)`, `(integer? x)`, `(complex? x)`, `(rational? x)`, `(boolean? x)`, `(symbol? x)`, `(pair? x)`, `(list? x)`, `(vector? x)`, `(port? x)`.

Dynamic types 2

A variable or function parameter may change type:

```
(set! id 2)
(set! id '#(2 3 4))
(set! id "333")
```

Generic functions can be defined.

```
;; x: string or symbol.
(define (my-length x)
  (string-length
    (if (symbol? x)
        (symbol->string x)
        x)))
```

Some functions are higher-order

Let $A =$`(-7 -5 -3 -2 11 13 17)`

Let $B =$`(2.3 hello 4 "allo")`

`(map abs A)` $\Rightarrow$ `(7 5 3 2 11 13 17)`

`(apply <= A)` $\Rightarrow$ `#t`

`map` and `apply` take a function has argument: they are higher-order functions.

The imperative profile of Scheme

The bang '!' signifies assignment (aka modification).

```
(set! x 2)
(set! x 3)
```

x $\Rightarrow$ 3

Let $A =$(1 2 3)

```
(set-car! A -2)
```

A $\Rightarrow$ (-2 2 3)

```
(set-cdr! A '(4 5))
```

A $\Rightarrow$ (-2 4 5)

and

`and` is a macro (aka special form), not a function.

It evaluates sequentially its arguments. If one of them is `#f`, it returns `#f`. If not, it returns the value of the last arguments.

`(and 1 2 3)` $\Rightarrow$ 3.

Useful to guard the evaluation of an expression.

```
(and (pair? L) (car L))
```

This is `#f` or the head of `L`.

or

`or` is a special form; it is not a function.

It evaluates sequentially its arguments. If one of them is not `#f`, it returns that value. If not, it returns `#f`.

```
(or (<= x y) (< x 0))
```

As for `and`, `or` may be applied to any type.

`(or 1 2 3)` $\Rightarrow 1$

cond

`cond` is a macro.

```
(define (o->str o)
  (cond
    ((string? o) o)
    ((number? o) (number->string o))
    ((symbol? o) (symbol->string o))
    (else #f)))
```

The conditions are tested in sequence.

For the first true one, the body of the clause is evaluated.

case

`case` is a macro.

```
(case key
  ((taf metar) (p-taf-metar))
  ((synop)     (p-synop))
  (else #f))
```

The key is compared (`eqv?`) with the list of contants.

The clause that has a constant `eqv?` to the key is evaluated.

Let

`let` is a macro.

A `let` binds a series of variables to values:

```
(let ((x  (expt 2 23))
      (pi (* 2 (asin 1))))
  ... x and pi are locals ...
  )
```

Let is a hidden lambda

Let is a macro defined using one lambda.

```
(let ((x  2)
      (y  3))
  e1 ... en)
```

is equivalent to

```
((lambda (x y) e1 ... en) 2 3)
```

Let*

`let*` allows sequential initializations.

```
(let* ((x  2)
       (y  (expt 3 x)))
  (cons x y))
```

$\Rightarrow$ (2 9)

Let* is a series of lambda(s)

`let*` is a macro defined using lambda(s).

```
(let* ((x 2)
       (y (expt 3 x)))
  (cons x y))
```

is translated into

```
((lambda (x)
  ((lambda (y) (cons x y))
    (expt 3 x))
  2))
```

Recursion 1

Encompasses iteration and all other forms of control flow.

```
(define (printl l)
  (if (pair? l)
      (begin
        (display (car l))
        (newline)
        (printl (cdr l)))))
```

`(printl (cdr l))` is a tail-call. It can be implemented as fast as an iteration.

Scheme: tail-call must not grow the frame stack.

Recursion 2

In Scheme all iterations are based on recursion.

```
(define (filter p l)
  (if (pair? l)
      (if (p (car l))
          (cons (car l) (filter p (cdr l)))
          (filter p (cdr l)))
      '()))
```

Filter can completely be tail recursive by defining a local function which accumulates the result.

Recursion 3

```
;; I: p, a function of one argument.
;;    ls, [e].
;; O: #t iff p(e), e in ls.
(define (for-all p ls)
  (if (pair? ls)
      (and (p (car ls)) (for-all p (cdr ls)))
      #t))
```

`(for-all p l)` is true iff all elements of `L` satisfy `p`.

Named let

A named let allows terse recursive loops.

```
(define (filter p l)
  (let loop ((ls l) (r '()))
    (if (pair? ls)
        (if (p (car ls))
            (loop (cdr ls) (cons (car ls) r))
            (loop (cdr ls) r))
        r)))
```

The `(let loop ...)` is akin to a function definition.

Variable arity of some functions

`(+ 1 2 3 4)` $\Rightarrow$ `10`

`(<= x y z w)` $\Rightarrow$ `#t` if `x`, `y`, `z`, `w` are in increasing order .

`(max 1 2 3)` $\Rightarrow$ `3`

`(min 1 2 3 4 5 6)` $\Rightarrow$ `1`

Variable arity, rest parm

Variable arity functions can be defined using a *rest parm*.

```
(define (f x y . L)
  (list x y L))
```

`L` is a *rest parm*.

Inside `f`, `L` is a list – could be empty.

`(f 1 2 3 4 5)` $\Rightarrow$ `(1 2 (3 4 5))`

`(f 1 2)` $\Rightarrow$ `(1 2 ())`

`(f 1)` $\Rightarrow$ error

For-each

Iterating over a list is a common task.

`(for-each procedure list)` $\Rightarrow$ `#<void>`

Done for its side-effect.

```
(define (display-ln . l)
    (for-each display l)
    (newline))
```

```
(display-ln "For " x " the result is " (sqrt x))
```

Instead of four calls to display and a call to newline.

Apply

The `apply` primitive applies a function to a list of arguments.

```
(define (f x y) ... )
```

```
(apply f '(1 2))
```

Let `L` = `(10 10 20)`

`(apply * L)` $\Rightarrow$ 2000

apply is of variable arity

It is impossible to dynamically discover the arity of a procedure.

A complex example of apply

```
;; I: p, a function of n arguments.
;;    ls, a list of n>0 lists of same length.
;; O: #t iff for all (p a1 ... an).
;;
(define (all p . ls)
  (if (and (pair? ls) (for-all pair? ls))
      (and (apply p (map car ls))
           (apply all p (map cdr ls)))
      #t))

(define (good? x y z) (<= x y z))

(all good? '(1 2 3) '(4 5 6) '(5 6 7))

⇒ #t
```

Input primitives 1

`(read)`: can read all basic data types – except procedure: vector, list, number, string, symbol, etc.

It can read complex structures like:

```
(area
   (bounding-box 50 100W 42 -64)
   (products (METAR)))
```

`(read-char)`: reads one character

A port can be specified, otherwise it is the current input port.

Input primitives 2

`(current-input-port)` ⇒ `#<input-port (stdin)>`

`(open-input-file "data")` ⇒ `#<input-port "data">`

Or the convenient form where the current input is temporarily redirected.

```
(with-input-from-file "data"
  (lambda ()
     ...
     (read)
      ...))
```

Input primitives 3

`read` and `read-char` may return eof object.

`(eof-object? x)` $\Rightarrow$ `#t` iff `x` is the eof-object.

```
(define (read-line port)
  (let loop ((r '()) (c (read-char port)))
    (if (not (eof-object? c))
        (if (char=? c #\newline)
            (list->string (reverse r))
            (loop (cons c r) (read-char port)))
        (list->string (reverse r)))))
```

`(peek-char)`: peek at the next character

Input primitives 4

Reading all the Scheme data from a file into a list.

```
(define (read-file f)
  (with-input-from-file f
    (lambda ()
      (let loop ((o (read)) (r '()))
        (if (eof-object? o)
            (reverse r)
            (loop (read) (cons o r)))))))
```

Note: the data must conform to Scheme syntax.

Output primitives, 1

`(current-output-port)` ⇒ `#<output-port (stdout)>`

`(open-output-file "data")` ⇒ `#<output-port>`

`(display` *exp port*`)`: writes exp to port.

`(write` *exp port*`)`: writes “as is” (to be `read` later).

`(write-char` *char port*`)`: writes one character.

port is always optional.

Output primitives, 2

Let $x =$ `#(1 2 3)`

Let $s =$ `"Hello"`

`(display s)` $\Rightarrow$ `Hello`

`(display x)` $\Rightarrow$ `#(1 2 3)`

`(write s)` $\Rightarrow$ `"Hello"`

`(write x)` $\Rightarrow$ `#(1 2 3)`

There is no “fprintf”. But, some implementations provide it.

Output primitives, 3

`(open-output-file "data")` ⇒ `#<output-port>`

Or the convenient form:

```
(with-output-to-file "data"
  (lambda ()
     ...
     (display result)
      ...))
```

The lambda is called with current output port redirected.

Closure, 1

A closure is a generalization of a function.

A closure is a function with private variables that can be modified.

```
(let ((x 2)
      (y 3))
  (lambda (z) (+ x y z)))
```

$\Rightarrow$ a procedure of one argument with x and y bound to 2 and 3.

Encapsulation of state

Closure can encapsulates state.

```
(define (make-counter init)
  (let ((count init))
    (lambda (x)
      (set! count (+ count x))
      count))
```

```
(define c1 (make-counter 1))
(define c2 (make-counter 1))
```

`(c1 1)` $\Rightarrow$ 2; `(c1 10)` $\Rightarrow$ 12; `(c1 3)` $\Rightarrow$ 15

`(c2 -1)` $\Rightarrow$ 0; `(c1 2)` $\Rightarrow$ 17

Count is local: `count` $\Rightarrow$ error

Functional programming 1

An objective is to make programming closer to mathematic.

Mathematic: static formulas, as in $E = mc^2$, $F = ma$. Substitution rule works in mathematic.

Programming: dynamic states.

```
(set! x 2)
(display x)
```

A very simple way to say it, functional programming is: try to avoid '!' !

It should be easier to reason about the program.

Functional programming 2

$$
\begin{aligned}
(\mathsf{max}\, x) &= x \\
(\mathsf{max}\, x_1 x_2) &= \text{if } x_1 > x_2 \text{ then } x_1 \text{ else } x_2 \\
(\mathsf{max}\, x_1 x_2 \ldots x_n) &= (\mathsf{max}\, x_1 (\mathsf{max}\, x_2 \ldots x_n))
\end{aligned}
$$

May be perceived as iteration and recursion:

```
;; l must be non empty.
(define (max . l)
   (let loop ((ls (cdr l)) (m (car l)))
      (if (pair? ls)
          (if (> (car ls) m)
              (loop (cdr ls) (car ls))
              (loop (cdr ls) m))
          m)))
```

Max

Closer to the mathematical definition:

```
(define (max . l)

  (define (max2 a b) (if (< a b) b a))

  (if (and (pair? l) (pair? (cdr l)))
      (max2 (car l) (apply max (cdr l)))
      (car l)))
```

Foldr

(foldr $f\ a\ (x_1\ x_2 \ldots x_n)) = (f\ x_1\ (f\ x_2\ \ldots (f\ x_n\ a) \ldots))$

A direct recursive approach:

```
(define (foldr f a l)
  (if (pair? l)
      (f (car l) (foldr f a (cdr l)))
      a))
```

`(foldr + 0 '(1 2 3))` is $1 + 2 + 3 \Rightarrow 6$

OO programming

Object Oriented programming can be done using closures.

```
(define (make-object ...)
  (let ((field1 ...)     ;; init
        (field2 ...)     ;; init
        ...
        (fieldn ...))

    (define (method1-private ...) ...)

    (lambda (mesg . parms)
      (case mesg
         ((add)   ...)
         ((reset) ...)
         ...))))
```

Communication with the object is done using the lambda and appropriate messages.

In Metcast server

fastcgi output is buffered, the buffer is local and all counters are hidden away in a let.

```
(define SRV:send-reset     #f)
(define SRV:send-reply     #f)
...
(let* ((b-len         (* 8 1024))
       (b-index       0)
       (b-max-chunks  (* 4 1024))
       (b             (make-vector b-max-chunks #f))
       (bp            0)
       (oport         (current-output-port))
       (output?       #f))


  (set! SRV:send-reset
        (lambda ()
          (set! output? #f) (set! b-index 0) (set! bp 0)
          (set! oport (current-output-port))))

  (set! SRV:send-reply
        (lambda fragments
         ...
        ))
  ...)
```

In Metcast decoders

The decoders buffer their output before uploading into the database.

```
(define (make-upload-buffer name)
  (let ((buffer-name      #f)
        (buffer-port      #f)
        (records-counter  0))

    (define (prepare!)
      (if (not (initialized?))
        (set! buffer-name (OS:tmpnam))
        (set! buffer-port (open-output-file buffer-name))
        (set! records-counter  0)
        (assert (initialized?))))

    (lambda (selector)
      (case selector
        ((add-record!)
         (prepare!)
         (++! records-counter)
         buffer-port))
        ... ))
```

Continuations 1

A continuation is a closure of one parameter with a stack.

It is created by capturing a point of execution with the current stack. Used to implement exceptions, debugger, co-routines, etc.

```
(define jump-in #f)

(let ((x 1))
  (call-with-current-continuation
    (lambda (k) (set! jump-in k)))
  (set! x (+ x 1))
  x)
```

$\Rightarrow$ 2

`(jump-in #f)` $\Rightarrow$ 3

`(jump-in #f)` $\Rightarrow$ 4

Continuations 2, exception

Error returns to `(display "Starting...")`.

```
(define throw #f)
(define (error msg) (display msg) (throw #f))

(define (process a)
  (if (number? a)
      (sqrt a)
      (error "Not a number.")))

(define (calculator)
 (call-with-current-continuation
   (lambda (k) (set! throw k)))

 (display "Starting...")(newline)
 (let loop ((s 0))
   (display s)(newline)
   (display "number>")
   (loop (+ s (process (read))))))
```

Continuations 3, a return-esc mechanism

```
(define return-esc #f)

(define (call-with-return-esc p . l)
 (call-with-current-continuation
   (lambda (k)
     (set! return-esc k)
     (apply p l))))

;; I: l, (x1 x2 ... xn)
;; O: x1*x2*...*xn
;;
(define (multiply l)
  (if (pair? l)
      (if (= (car l) 0)
          (return-esc 0)
          (* (car l) (multiply (cdr l))))
      1))

(call-with-return-esc multiply '(1 0 4 5)) ⇒ 0
```

Because of `return-esc 0`, it does not unstack the calls, but jumps back after `call-with-return-esc`.